



## P2P Join Query Processing over Data Streams

Wenceslao Palma, Reza Akbarinia, Esther Pacitti, Patrick Valduriez

### ► To cite this version:

Wenceslao Palma, Reza Akbarinia, Esther Pacitti, Patrick Valduriez. P2P Join Query Processing over Data Streams. BDA: Bases de Données Avancées, Oct 2009, Namur, Belgium. inria-00416819

**HAL Id: inria-00416819**

**<https://inria.hal.science/inria-00416819>**

Submitted on 15 Sep 2009

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# P2P Join Query Processing over Data Streams

Wenceslao Palma <sup>†</sup>                      Reza Akbarinia <sup>†</sup>  
Wenceslao.Palma@univ-nantes.fr      Reza.Akbarinia@inria.fr

Esther Pacitti <sup>‡</sup>                      Patrick Valduriez <sup>‡</sup>  
pacitti@lirmm.fr                      Patrick.Valduriez@inria.fr

INRIA and LINA - Université de Nantes <sup>†</sup>  
2 Rue de la Houssinière, 44322 Nantes - FRANCE

INRIA and LIRMM - Université de Montpellier 2 <sup>‡</sup>  
161 Rue Ada, 34392 Montpellier Cedex 5 - FRANCE

## Abstract

*De nombreuses applications distribuées partagent la même nécessité de traiter des flux de données de façon continue, par ex. la surveillance de réseau ou la gestion de réseaux de capteurs. Dans ce contexte, un problème important et difficile concerne le traitement de requêtes continues de jointure qui nécessite de maintenir une fenêtre glissante sur les données la plus grande possible, afin de produire le plus possible de résultats probants. Dans cet article, nous proposons une nouvelle méthode pair-à-pair, DHTJoin, qui tire parti d'une Table de Hachage Distribuée (DHT) pour augmenter la taille de la fenêtre glissante en partitionnant les flux sur un grand nombre de noeuds. Contrairement aux solutions concurrentes qui indexent tout les tuples des flux, DHTJoin n'indexe que les tuples requis pour les requêtes et exploite, de façon complémentaire, la dissémination de requêtes. DHTJoin traite aussi le problème de la dynamique des noeuds, qui peuvent quitter le système ou tomber en panne pendant l'exécution. Notre évaluation de performances montre que DHTJoin apporte une réduction importante du trafic réseau, par rapport aux méthodes concurrentes.*

**Mots-clefs:** Flux de données, Requêtes de jointure, Table de hachage distribuée.

## 1. Introduction

Recent years have witnessed major research interest in Data Stream Management Systems (DSMS), which can manage continuous and unbounded sequences of data items. There are many applications that generate data streams including financial applications [7], network monitoring [28], telecommunication data management [6], sensor networks [4], etc. Processing

a query over a data stream involves running the query continuously over the data stream and generating a new answer each time a new data item arrives. However, the unbounded nature of data streams makes it impossible to store the data entirely in bounded memory. This makes difficult the processing of queries that need to compare each new arriving data with past ones. For example, real data traces of IP packets from an AT&T data source [11] show an average data rate of approximately 400 Mbits/sec, which makes it hard to keep pace for a DSMS. Moreover, a DSMS may have to process hundreds of user queries over multiple data sources. For most distributed streaming applications, the naive solution of collecting all data at a single node is simply not viable [8]. Therefore, we are interested in techniques for processing continuous queries over collections of distributed data streams. This setting imposes high processing and memory requirements. However, approximate answers are often sufficient when the goal of a query is to understand trends and making decisions about measurements or utilizations patterns.

One technique for producing an approximate answer to a continuous query is to execute the query over a sliding window [12] that maintains a restricted number of recent data items. This allows queries to be executed in finite memory, in an incremental manner by generating new answers each time a new data item arrives. Moreover, in the majority of real world applications emphasizing recent data is more informative and useful than old data. Notice that a sliding window is a natural method for approximation that is part of the query semantics expressed by the user in the query. The size of a window is specified using either a time interval (time-based) or a count on the number of tuples (count-based). In this work we consider time-based windows.

In continuous query processing the join operator is one of the most important operators, which can be used to detect trends between different data streams. For example, consider a network monitoring application that needs to issue a join query over traffic traces from various links, in order to monitor the total traffic that passes through three routers ( $R_1$ ,  $R_2$  and  $R_3$ ) and has the same destination host within the last 10 minutes. Data collected from the routers generate streams  $S_1, S_2$  and  $S_3$ . The content of each stream tuple contains a packet destination, the packet size and possibly other information. This query can be posed using a declarative language such as CQL [2], a relational query language for data streams, as follows:

```

 $q_1$ : Select sum ( $S_1.size$ )
      From  $S_1[range\ 10\ min], S_2[range\ 10\ min], S_3[range\ 10\ min]$ 
      Where  $S_1.dest=S_2.dest$  and  $S_2.dest=S_3.dest$ 

```

To emphasize access to recent data, the window conceptually slides over the input streams thereby giving rise to a type of join called *sliding window join*. In this paper, we address the problem of computing approximate answers to sliding window joins over data streams. Our solution involves a scalable distributed sliding window that takes advantage of the indexing power of DHT networks and can be equivalent to thousands of centralized sliding windows. We propose a method, called DHTJoin, which deals with efficient processing of join queries over all data items which are stored in the distributed sliding window. To this end, DHTJoin combines hash-based placement of tuples in the DHT and dissemination of queries. We evaluated the performance of DHTJoin through simulation. The results show the effectiveness of our solution compared with previous work.

This paper is an extended version of [23] with the following added value. First, we present

a dissemination system (Section 3.1) based on the trees formed by DHT links that uses  $O(n - 1)$  messages. This yields an important reduction of network traffic compared with the  $O(\frac{n \log n}{2})$  messages generated by the dissemination system proposed in our previous work. Considering that nodes that fail (or leave the network) during query execution may cause problems in the generation of join results and the dissemination of queries, we propose a solution to deal with node failures (Section 4). In Section 5, we show analytically what is the number of nodes that allows achieving a certain degree of completeness given a continuous join query. Finally, in Section 6 we report experimental results that show the effectiveness of our approach.

### 1.1. Contributions

In summary, we propose a novel method (DHTJoin) for the execution of continuous join queries with the following contributions: (1) DHTJoin identifies, using query predicates, a subset of tuples in order to index the data required by the user’s queries, thus reducing network traffic. This is more efficient than the approaches based on structured P2P overlays, e.g. PIER [13] and RJoin [14], which typically index all tuples in the network. Furthermore, our approach dynamically indexes tuples based on new attributes when new submitted queries contain different predicates, (2) we provide an analytical evaluation of the best number of nodes to obtain a certain degree of completeness given a continuous join query, (3) DHTJoin tackles the dynamic behavior of DHT networks during query execution and dissemination of queries. When nodes fail during query dissemination, DHTJoin uses a gossip-based protocol that assures 100% of network coverage. When nodes fail during query execution, DHTJoin propagates messages to prevent nodes of sending intermediate results that do not contribute to join results, thereby reducing network traffic.

### 1.2. Organization

The rest of this paper is organized as follows. In Section 2, we introduce our system model and define the problem. In Section 3, we describe DHTJoin. In Section 4, we discuss how DHTJoin deals with node failures. In Section 5, we provide an analysis of result completeness of our algorithms which relates memory constraints, stream arrival rates and result completeness. In Section 6, we provide a performance evaluation of our solution through simulation using Java. In Section 7, we discuss related work. Section 8 concludes.

## 2. System Model and Problem Definition

In this section, we introduce a general system model for processing data streams over DHTs, with a DHT and data model, and a stream processing model. Then, we state the problem.

### 2.1. DHT and Data Model

In our system, the nodes of the overlay network are organized using a DHT protocol. While there are significant implementation differences between DHTs [24][27], they all map a given key  $k$  onto a node  $p$  using a hash function and can lookup  $p$  efficiently, usually in

$O(\log n)$  routing hops where  $n$  is the number of nodes. DHTs typically provide two basic operations:  $put(k, data)$  stores a key  $k$  and its associated  $data$  in the DHT using some hash function;  $get(k)$  retrieves the data associated with  $k$  in the DHT. In a DHT each node has a identifier denoted by  $node_{id}$ . Nodes insert data in the form of relational tuples and queries are represented in a relational query language for data streams such as CQL [2]. Tuples belonging to the same stream are inserted by the same node and continuous queries are originated at any node of the network. Tuples and queries are timestamped to represent the time that they are inserted in the network by some node. We assume that data and query sources are equipped with well-synchronized clocks by using the public domain Network Time Protocol (NTP) designed to work over packet-switched and variable latency data networks and already tested in distributed DSMS [29]. Additionally, each query is associated with a unique key  $q_{id}$  used to identify it in query processing, optimization tasks and to relate it to the node that submitted it.

Let us now formally define continuous join queries and the type of continuous queries that we consider in our approach. Let  $S = \{S_1, S_2, \dots, S_m\}$  be a set of data streams. Each data stream  $S_i$  has a relational schema  $(A_1^i, A_2^i, \dots, A_{n_i}^i)$ , where each  $A_j^i$  is an attribute. We use equijoin and conjunctive predicates, i.e., the *where* clause uses exclusively conjunctions of atomic equality conditions. Let  $Q_i = (S', \mathcal{P})$  be a continuous join query defined over  $S' \subseteq S$  and composed by  $\mathcal{P}$  that represents a set of equijoin predicates. As in [35][14], we identify two types of join queries depending on the attributes involved in  $\mathcal{P}$ . A query of type 1 is a join query with a set of equijoin predicates as following:  $\mathcal{P} = \{(S_1.A_k^1 = S_2.A_k^2), (S_2.A_k^2 = S_3.A_k^3), \dots, (S_{m-1}.A_k^{m-1} = S_m.A_k^m)\}$ , i.e., the join attribute is the same in all the relations of the query (e.g. query  $q_1$  of Sect. 1). A query of type 2 is a join query with a set of equijoin predicates as following:  $\mathcal{P} = \{(S_1.A_k^1 = S_2.A_l^2), (S_2.A_l^2 = S_3.A_l^3), (S_3.A_m^3 = S_4.A_m^4), \dots, (S_{m-1}.A_{n_m}^{m-1} = S_m.A_{n_m}^m)\}$ , i.e., the join attributes are different and adjacent joins must have a common relation.

## 2.2. Stream Processing Model

A data stream  $S_i$  is a sequence of tuples ordered by an increasing timestamp where  $i \in [1..m]$  and  $m \geq 2$  denotes the number of input streams. At each time unit, a number of tuples of average size  $l_i$  arrives to stream  $S_i$ . We use  $\lambda_i$  to denote the average arrival rate of a stream  $S_i$  in terms of tuples per second.

Many applications are interested in making decisions over recently observed tuples of the streams. This is why we maintain each tuple only for a limited time. This leads to a sliding window  $S[W_i]$  over  $S_i$  that is defined as follows. Let  $W_i$  denotes the size of  $S[W_i]$  in terms of seconds, i.e. the maximum time that a tuple is maintained in  $S[W_i]$ . Let  $TS(s)$  be a function that denotes the arrival time of a tuple  $s$  and  $t$  be current time. Then  $S[W_i]$  is defined as  $S[W_i] = \{s | s \in S_i \wedge (t - TS(s) \leq W_i)\}$ . Tuples continuously arrive at each instant and expire after  $W_i$  time steps (time units). Thus, the tuples under consideration change over time as new tuples get added and old tuples get deleted. In practice, when arrival rates are high, the window sizes are long and the memory dedicated to the sliding window is limited, it becomes full rapidly and many tuples must be dropped before they naturally expire. In this case, we need to decide whether to admit or discard the arriving tuples and if admitted, which of the existing tuples to discard. This kind of decision is made using a load shedding strategy

[26][30] which yields that only a fraction of the complete result will be produced.

### 2.3. Problem Definition

In this paper, we address the problem of processing join queries over data streams. We view a data stream as a sequence of tuples ordered by monotonically increasing timestamps. The nodes are assumed to synchronize their clocks using the public domain Network Time Protocol (NTP), thus achieving accuracies within milliseconds [3]. Each tuple and query have a timestamp that may be either implicit, i.e. generated by the system at arrival time, or explicit, i.e. inserted by the source at creation time.

This paper focuses on query execution (not query optimization). Thus, we assume the existence of a query optimizer that translates a query represented in CQL [2] into a query plan in the form of an operator tree. Since an MJoin operator [31] is used by default to specify join operations, only the join order needs to be specified by the optimizer, i.e. the choice of how to execute MJoin operators (e.g. which nodes) is done at runtime using our method. Each query  $Q_i$  has a query plan  $QP_i$  that specifies the ordering of the join operations.

Formally, the problem can be defined as follows. Let  $S = \{S_1, S_2, \dots, S_m\}$  be a set of data streams, and  $QP = \{QP_1, QP_2, \dots, QP_n\}$  be a set of query plans of the following set of continuous join queries  $Q = \{Q_1, Q_2, \dots, Q_n\}$ , where  $Q_i = (S', \mathcal{P})$  is a continuous join query defined over  $S' \subseteq S$  and  $\mathcal{P}$  represents a set of equijoin predicates. Our goal is to provide an efficient method to execute  $QP$  over  $S$  in terms of network traffic.

## 3. DHTJoin Method

In this section, we describe our solution, DHTJoin, for processing continuous join queries using DHTs. DHTJoin has two steps: dissemination of queries and indexing of tuples. A query is disseminated using the embedded tree invariants to DHTs networks and a tuple inserted by a node is indexed, i.e., stored at another node using DHT primitives. However, a node indexes a tuple only if there is a query that contains an attribute of the arriving tuple in  $\mathcal{P}$ . To this end, a node stores locally a disseminated query and once it receives a tuple it checks for already disseminated queries that contain an attribute of the arriving tuple in  $\mathcal{P}$ .

We describe the design of DHTJoin based on Chord which is a simple and very popular DHT. However, the techniques used here can be adaptable to others DHTs such as Pastry [25] and Tapestry [33].

To process a query, we consider different kinds of nodes. The first kind is Stream Reception Peers (SRP) for indexing tuples to the second kind of nodes, the Stream Query Peers (SQP). In Figure 1(b), nodes 3, 6 and 7 correspond to SRP because they receive tuples belonging to streams  $z$ ,  $y$ , and  $x$  respectively. SQP are responsible for executing query predicates over the arriving tuples using their local sliding windows, and sending the results to the third kind of node(s), the User Query Peers (UQP). In Figure 1(b), nodes 1 and 4 are SQP because node 1 computes the join predicate  $X.B = Y.B$  of query  $q_2$  (submitted at node 0) and node 4 performs the join predicate  $Y.C = Z.C$  of  $q_2$ . In addition, node 0 is a UQP because query  $q_2$  was submitted at this node.

To support dissemination of queries, a node must be a dissemination node (i.e. executes a dissemination protocol) while to index tuples, a node must be a DHT peer. Note that the

difference between SRP, SQP and UQP is functional and the same node can support all these functionalities.

### 3.1. Disseminating Queries

Each new query issued by users should be disseminated to all nodes because by using  $S'$  and the set of predicates  $\mathcal{P}$  of a query a node decide which tuples and attributes should be indexed. The query dissemination system consists of a set of DHT nodes. A query can originate at any of the nodes and is disseminated using a tree [5].

To disseminate a query, DHTJoin dynamically builds a dissemination tree as proposed in [9]. The basic idea is to consider that in a DHT as Chord a *lookup* operation can be perceived as a binary search [9] that generates a binary tree using the nodes (links) stored in the routing table. The root of the tree is the node that submits the query (an UQP node). The query is disseminated from the root node to all nodes of the DHT using a divide-and-conquer approach. When a node receives a disseminated query, it is stored locally in a *query table* ( $QT$ ), thus allowing to know what is the attribute of an arriving tuple that must be used in the indexing process. This is important since a tuple  $s_i$  is indexed using an attribute  $A_j^i$  only if it is contained in the set  $\mathcal{P}$  allowing to decrease network traffic and providing a better utilization of local SQP resources by avoiding the indexing of tuples using an attribute that is not being involved in a query.

To disseminate a query, an UQP node creates a dissemination message  $Dmsg = (node_{id}, q_{id}, Q_i, QP_i, ts, \mathcal{R})$  containing its own node identifier  $node_{id}$ , an unique query identifier  $q_{id}$ , the query  $Q_i = (S', \mathcal{P})$ , the query plan  $QP_i$ , a timestamp  $ts$  that denotes the arrival time of  $Q_i$  and a range of dissemination  $\mathcal{R}$ . A node that receives a  $Dmsg$  store the query in its  $QT$  and creates a new  $Dmsg$  preserving the  $node_{id}$ , the  $q_{id}$ , the timestamp  $ts$ , the query  $Q_i$  and the query plan  $QP_i$ , and changing  $\mathcal{R}$ . For example, using a fully-populated Chord ring with 8 nodes, each one contains a routing table of  $\log(n)$  entries called fingers. The  $i^{th}$  entry in the table at node  $n$  contains the identity of the first node that succeeds or equal  $n + 2^i$ . A dissemination message initiated at node 0 is sent to finger nodes 1, 2 and 4 giving them the disseminations limits  $[1,2)$ ,  $[2,4)$  and  $[4,0)$  respectively. The disseminations limits are used to restrict the forwarding space of a node and they are constructed using as a upper bound the finger  $i + 1$ . Each node applies the same principle reducing the search scope. When node 2 receives the dissemination message with limits  $[2,4)$  it examines the routing table and sends the message to node 3. Once node 4 receives the dissemination message it examines the routing table and sends the message to nodes 5 and 6 with limits  $[5,6)$  and  $[6,0)$  respectively. In the same way, node 5 does not continues with the dissemination process (since there are no nodes between  $[5,6)$ ) and node 6 disseminates the message to node 7.

This forwarding process generates  $n - 1$  messages and a tree of depth  $\log(n)$ , which fixes the latency of query dissemination.

### 3.2. Indexing Tuples

The indexing of tuples allows DHTJoin to distribute the query workload across multiple DHT nodes. Let us describe how DHTJoin indexes tuples for streams  $S = \{S_1, S_2, \dots, S_m\}$ . Let  $s_i$  be a tuple belonging to  $S_i$ . Let  $A = (A_1^i, A_2^i, \dots, A_{n_i}^i)$  be the set of attributes in  $s_i$

and  $val(s_i, A_j^i)$  be a function that returns the value of the attribute  $A_j^i \in A$  in tuple  $s_i$ . Let  $h$  be a uniform hash function that hashes  $val(s_i, A_j^i)$  into a DHT key, i.e. a number which can be mapped to a  $node_{id}$ . A node that index a tuple  $s_i \in S_i$  creates a message  $Index = (S_i, s_i, A_j^i, ts)$  containing the stream  $S_i$  which the tuple belongs to, the tuple  $s_i$  being indexed, the attribute used to index the tuple and a timestamp  $ts$  that denotes the arrival time of the tuple. Let  $S[W_i]$  denote a sliding window on stream  $S_i$ . Recall that we use time-based sliding windows where  $W_i$  is the size of the window in time units. At time  $t$ , a tuple  $s_i$  belongs to  $S[W_i]$  if it has arrived in the time interval  $[t - W_i, t]$ .

For indexing a tuple  $s_i$  that arrives at an SRP, each tuple obtains an index key computed as  $key = h(val(s_i, A_j^i))$ . The attribute  $A_j^i$  in  $s_i$  is chosen by searching locally in the  $QT$  for queries that contains  $A_j^i$  in  $\mathcal{P}$ . Then to index  $s_i$  the SRP node creates a  $Index$  message and sends it to a SQP (the node responsible for  $key$  in the DHT), by performing  $put(key, Index)$ . Thus, tuples of different streams having the same  $key$  are put in the same SQP node and are stored in sliding windows where they are processed to produce the result of a specific join predicate.

### 3.3. Query Execution

Query processing in a DSMS entails the generation and execution of a query plan. This paper focuses on the execution part. For simplicity, we assume that the query plan is an operator tree that specifies the ordering of operations (i.e. join order) and it is included in the  $Dmsg$  message of the query dissemination step (see Section 3.1).

Queries of type 1 are executed using partitioned parallelism with SQP nodes implementing the MJoin operator [31]. A query plan contains an operator tree for each stream present in the query that could be optimized locally, thus generating a new operator tree. Each node in the operator tree represents a join operator and an edge represents the next stream to probe. Queries of type 2 are executed using pipelined parallelism (see Figure 2). For queries of type 2, the query plan is assumed to be generated by a centralized query optimizer based on a cost model which captures information regarding data (e.g. tuples' arrival rates) and operators (e.g. cost of a join) [35]. Each node in the operator tree represents a join operator implemented using MJoin and an edge represents the next step in the pipeline.

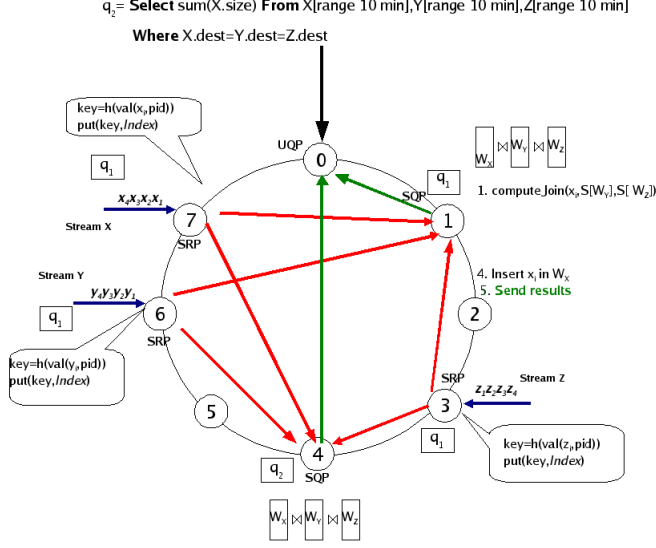
In this section, we describe the execution of queries of type 1 and 2 in DHTJoin.

#### 3.3.1. Queries of Type 1

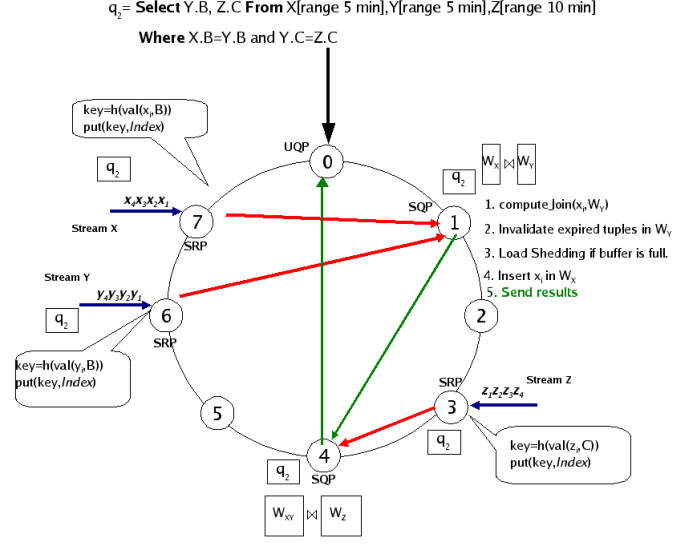
In this type of queries, DHTJoin uses partitioned parallelism [19] where different nodes execute independently the same query plan on different data partitions. By default, DHTJoin instantiates an MJoin operator [31] for queries of type 1. MJoin considers  $n$  inputs streams symmetrically and allows the tuples from the streams to arrive in an arbitrary interleaved fashion. The basic algorithm of MJoin creates as many hash tables (states) as there are join attributes in the query. When a new tuple from a stream arrives into the system, it is probed with the other  $n - 1$  streams in some order to find the matches for the tuple. The order in which the streams are probed is called the *probing sequence*.

Choosing a *probing sequence* is very important in MJoin because it must ensure that the smallest number of intermediate results is generated. This process is supported by heuristic-based ordering algorithms [12][31]. MJoin is very attractive when processing continuous





(a) A join example of query type 1



(b) A join example of query type 2

Figure 1: Query Processing in DHTJoin

queries over data streams because the query plans can be changed by simply changing the *probing sequence*. Thus, each SQP node that processes a query of type 1 can optimize the execution plan of the query independently. Let us illustrate how DHTJoin performs query processing with the following query of type 1:

$q_1$ : Select sum ( $X.size$ )  
 From  $X[range\ 10\ min], Y[range\ 10\ min], Z[range\ 10\ min]$   
 Where  $X.dest = Y.dest = Z.dest$

Query  $q_1$  is submitted at node 0 and disseminated, using the strategy proposed in Section 3.1, over the entire network as soon as it is submitted. SRP nodes 7, 6 and 3 index  $x_i$ ,  $y_i$  and  $z_i$  tuples and check locally in their  $QT$  whether  $q_1$  contains in  $\mathcal{P}$  an attribute belonging to the arriving tuples. Recall that in a query of type 1, the join attribute is the same in all relations, so that all the tuples having the same attribute value are located in the same SQP node without producing intermediate results. Therefore,  $q_1$  can be executed independently at different SQP nodes, each using an MJoin operator. In our example, SQP nodes 1 and 4 process  $q_1$  on different partitions of  $X$ ,  $Y$  and  $Z$  streams using an MJoin operator (see Figure 1(a)). The results produced by SQP nodes 1 and 4 are sent directly to the UQP node (whose address was provided in the  $Dmsg$  message when  $q_1$  was disseminated).

### 3.3.2. Queries of Type 2

DHTJoin executes queries of type 2 using pipelined parallelism [19] where different nodes run in a pipelined fashion such that tuples output by a node can be fed to another node as they get produced. Recall that DHTJoin partitions the streams by hash functions. For example, let us consider query  $q_2$  with the following set of predicates  $\{(X.B = Y.B), (Y.C = Z.C)\}$ . Streams  $X$  and  $Y$  are indexed based on the value of attribute  $B$  while stream  $Z$  is indexed based on the value of attribute  $C$  which is placed at a node different from where the stream

$Y$  is indexed. Therefore, redirection of intermediate join results is necessary in this type of query. Another solution is to index the stream  $Y$  twice, i.e. based on attributes  $B$  and  $C$  executing  $X \bowtie_B Y$  and  $Y \bowtie_C Z$  in parallel. However, we do not consider this solution for the two following reasons: (1) It duplicates unnecessarily the indexing of  $Y$  tuples and (2) It introduces more messages and processing costs because the output tuples of the two joins must be processed to find the final join result.

For queries of type 2, we assume that the query optimizer generates a query plan based on a bushy tree of binary joins that has the potential of executing independents subtrees concurrently. Local operators are executed using an MJoin operator and can be optimized as for queries of type 1. Let us illustrate how DHTJoin performs query processing using the following query of type 2:

$q_2$ : Select Y.B, Z.C  
 From X[range 5 min], Y[range 5 min], Z[range 5 min]  
 Where X.B=Y.B and Y.C=Z.C

This query specifies an equijoin among  $X$ ,  $Y$  and  $Z$  streams over the last 5 minutes. Query  $q_2$  is submitted at node 0 and disseminated over the entire network as soon as it is submitted. Thus, after a while, all nodes know the existence of this query and are able to index the incoming streams (tuples). We assume that the query plan generated for  $q_2$  is  $(X \bowtie_B Y) \bowtie_C Z$ . Once an  $X$ -,  $Y$ - or  $Z$ -tuple arrives at nodes 7, 6 and 3 respectively, each node checks locally in its  $QT$  whether the query  $q_2$  contains in  $\mathcal{P}$  an attribute belonging to the arriving tuple (see Figure 1(b)). If so, nodes 7, 6 and 3 execute the task of an SRP. For instance, in our example, node 7 indexes  $x_i$  because the attribute  $B \in X$  is in the set  $\mathcal{P}$  of  $q_2$ . Node 7 creates a message  $Index = (X, x_i, B, ts)$ , generates an index key using  $key = h(val(x_i, B))$  and indexes the tuple using  $put(key, Index)$ . The equijoin predicate  $X.B = Y.B$  belonging to  $q_2$  is evaluated at a SQP (node 1) only with tuples that arrive in the system after the query.

Sliding windows are used at each SQP node, as for queries of type 1, as follows. For example, at node 1 in Figure 1(b), tuples expired in  $S[W_Y]$  are invalidated upon the arrival of  $X$ -tuples. The load shedding procedure is executed over  $S[W_X]$ 's buffer if there is not enough memory space to insert the arriving tuple.

The SQP node 1 searches in the query plan of  $q_2$  what is the next step to follow and concludes that the intermediate results  $x_i y_j$  must be sent to another node using the value of  $C$  attribute belonging to the  $Y$ -tuple. Thus the SQP node 1 creates a message  $Index = (XY, x_i y_j, C, max(TS(x_i), TS(y_j)))$ , generates an index key using  $key = h(val(y_j, C))$  and index the intermediate tuple using  $put(key, Index)$  to SQP node 4. The join result tuples produced by SQP node 4 are immediately sent to the appropriate UQP node (whose address is provided when starting query dissemination).

## 4. Dealing with Node Failures

In this section, we discuss how DHTJoin deals with node failures during query execution. By node failure, we mean various situations by which a DHT node stops participating in query execution (e.g. because it crashes). We address this issue considering two situations: (1) *Failure of a node during query dissemination*. Recall that the dissemination of queries allows to decrease network traffic by avoiding the indexing of tuples using an attribute that

is not being involved in a query. However, its benefits can be lost when the tree hierarchical organization of the dissemination is broken due to node failures. (2) *Failure of a node during query execution*. The failure of a node stops the indexing of tuples. With queries of type 2, this situation can generate partial results that never contribute to generate join results.

#### 4.1. Failures during Query Dissemination

In DHTJoin, continuous join queries are originated at any node of the DHT and disseminated using a tree. The dissemination of queries achieves a network coverage of 100%, takes  $O(\log n)$  hops to reach every node in the network and generates  $n - 1$  messages. However, dynamic changes of the structure of the DHT network can disturb the dissemination. The failure of a node in the tree structure generated by the dissemination makes the entire subtree under this node unreachable. To provide reliability in the dissemination of queries, we propose to use a gossip based protocol as a complementary to our tree based dissemination.

Basically, gossip proceeds as follows: a node  $n_i$  knows a group of other nodes or *contacts*, which are maintained in a list called  $n_i$ 's *view*. Periodically  $node_i$  selects a contact  $node_j$  from its view to gossip:  $node_i$  sends its information to  $node_j$  and receives back other information from  $node_j$ .

We integrate gossip to DHTJoin's dissemination procedure as follows. The *view* maintained by the nodes is the neighbor list present in DHTs. All nodes that receive a disseminated query forward periodically the query to a randomly chosen neighbor. To this end, a node creates a gossip message  $Gmsg$  and executes  $send(receiver, Gmsg)$  where *receiver* is the destination node of message  $Gmsg$ .

Our algorithm to gossip query dissemination messages proceeds as follows. A message  $Gmsg$  is generated at any node that has already received a user-level query  $Q_i$ . A message  $Gmsg = (Q_i, q_{id}, QP_i, TS(Q_i), L_d)$  contains a query  $Q_i$ , a unique query identifier  $q_{id}$ , a query plan  $QP_i$ , a timestamp  $TS(Q_i)$  and a partial dissemination list  $L_d$  composed by the nodes of its local *view* to which the message has been sent (not necessarily received by all nodes of  $L_d$  due to the dynamic nature of the network) and the node that sent the message to it. To process a gossip message, a node that receives a message  $Gmsg = (Q_i, q_{id}, QP_i, TS(Q_i), L_d)$  chooses a random node  $n_r$  from its *view* and forwards the message  $(Q_i, q_{id}, QP_i, TS(Q_i), L_d \cup n_r)$  to  $n_r$  only if it has not been already chosen in previous gossip rounds.

#### 4.2. Failures during Query Execution

DHTJoin distributes the query workload across multiple DHT nodes and provides a mechanism that avoids indexing tuples using attributes not contained in the set  $\mathcal{P}$  of a query. However, when a node fails, another node can generate partial results irrespective of whether they produce join query results. In this section, we address the problem of indexing partial results that never contribute to generate join results.

For example, let us consider the following query plan  $(V \bowtie W) \bowtie ((X \bowtie Y) \bowtie Z)$  for a query of type 2 where there are nodes connected by a producer-consumer relationship, whereby a producer node generates tuples to be processed by a consumer node [32]. The query plan (see Figure 2) shows the relations between producers and consumers. We assume that a join operator  $Op_i$  resides at node  $n_i$ . Operator  $Op_3$  is a producer of  $X \bowtie Y \bowtie Z$  tuples

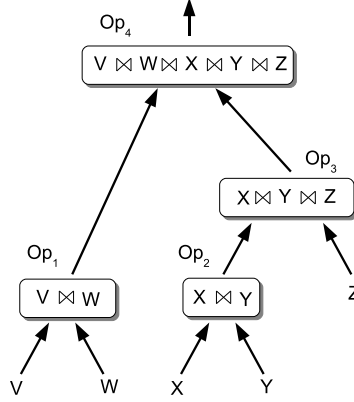


Figure 2: Query Plan of a 5-way continuous join query of type 2

for  $Op_4$  and a consumer w.r.t  $Op_2$  and SRP of stream  $Z$ . Recall that in a query of type 2, the operators are placed at different SQP nodes and the query plan is provided in the query dissemination step. If the node  $n_1$  fails, the indexing of  $V \bowtie W$  intermediate result tuples is stopped, thus yielding no join results because of no matching tuples in node  $n_4$ . Furthermore, if no matching tuple of  $V \bowtie W$  appears at node  $n_4$  before expiration of  $X \bowtie Y \bowtie Z$  tuples, the resources involved in sending, processing and storing these tuples are wasted.

To address this problem, we propose the following solution. If node  $n_4$ , where  $Op_4$  is executed, detects that  $V \bowtie W$  tuples are not being generated by node  $n_1$  it sends a message to node  $n_3$  to alert that it is not necessary to send  $X \bowtie Y \bowtie Z$  tuples. Consequently, as the demand of  $Op_3$  as a consumer has changed, it propagates the alerting message to node  $n_2$  and to the SRP of stream  $Z$  only if there does not exist another query that needs  $X \bowtie Y \bowtie Z$  tuples generated by  $Op_3$ . This condition is verified at all the operators that receive an alerting message. Once the communication with  $n_1$  is established again, node  $n_4$  sends a resume message to  $n_3$  in order to continue with the production of tuples and node  $n_3$  propagates the resume message it proceeds. If in a query plan, a consumer also acts as a producer, it is not necessary to alert its consumer. The reason is that a consumer is always testing its producers in the query plan in order to detect a problem. Therefore, the consumer that detects that there are tuples not being generated by a producer must trigger an alert message only to the other producers (the descendents) in the query plan if any. Procedure 1 describes the behaviour of the consumer that trigger the alert message to the producers of the query plan. Procedure 2 describes the behaviour of a producer in order to handle and alert message.

---

**Procedure 1** Send\_AlertMSG( $q$ )

---

**Input:** the query  $q$

- 1: **for** all the descendents  $\in$  query plan of  $q$  **do**
  - 2:    $alertMSG \leftarrow \{q, \{suspend|resume\}\}$
  - 3:   send(myID, alertMSG)
  - 4: **end for**
- 

In Procedure 1, a consumer sends an alert message to all the other producers of the query

plan of query  $q$ . The consumer sends a suspend message when it detects that there are tuples not being generated by a producer. Otherwise, it sends a resume message.

---

**Procedure 2** Handle\_AlertMSG(consumerID, alertMSG)

---

**Input:** **consumerID**, the identifier of the consumer node in the Chord ring. **alertMSG** is a message containing the identification of the query  $q$  and the type of action {suspend, resume}

- 1: **if** notExists( $q_i \in QT \neq q$ ) **then**
- 2:   propagate\_AlertMSG(myID, alertMSG);
- 3: **end if**
- 4: **if** (action is suspend) **then**
- 5:   suspend( $q$ )
- 6: **else**
- 7:   resume( $q$ )
- 8: **end if**

---

In Procedure 2, Line 1 verifies that there does not exist another query in  $QT$  that needs the tuples generated by the producer that receives the message. If so, the producer acting as a consumer sends the message to its descendents (Line 2) in the query plan of  $q$ . Finally, the producer performs appropriate operations to suspend (Line 5) or reactivate (Line 7) locally the production of tuples related to  $q$ . By eliminating unnecessary intermediate results, this optimization yields an important reduction of network traffic and a better utilization of local resources.

## 5. Analysis of Result Completeness

The notion of result completeness is important in distributed and P2P databases since partial (incomplete) query answers are often only possible [22][16]. Result completeness is thus defined as the fraction of results actually produced over the total results (which could be produced under perfect conditions). In data streaming applications, the potential high arrival rates of streams impose high processing and memory requirements. However, approximate answers are often sufficient when the goal of a query is to understand trends and making decisions about measurement or utilization patterns. Query approximation can be done by limiting the size of states maintained for queries [15]. In our analysis we focus in the case where the memory allocated to maintain the state of a query is not sufficient to keep the window size entirely, thus reducing the received join results and completeness. DHTJoin provides more memory to store tuples, but we consider that determining the number of computing resources necessities to achieve a certain degree of completeness for a given query is an important aspect in the setup phase of DHTJoin.

In this section, we propose formulas which relate peer memory constraints, stream arrival rates, and result completeness. We will use these formulas in our performance evaluation and they could be useful to a DHTJoin user (e.g. an application developer) to define and tune a DHT network for specific application requirements. We provide the necessary equations to calculate the completeness in a 2-way join and afterwards we generalize our results for a  $m$ -way join.

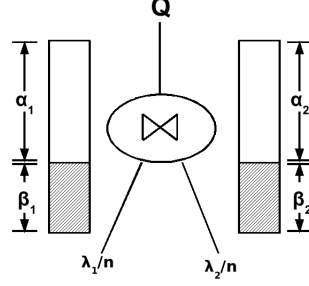


Figure 3: A join state including stored and non stored tuples

For ease of analysis, we make simplifying assumptions: the tuples are uniformly distributed across the DHT network; the memory assigned to store tuples is the same at each peer; we use the average rate to characterize the rate of arrivals of incoming tuples and stream tuples arrive in monotonically increasing order of their timestamps. In order to illustrate our analysis, let us consider the following join query over two streams  $S_1$  and  $S_2$ :

$Q$ : Select \*  
 from  $S_1$ [range 5 min],  $S_2$ [range 5 min]  
 where  $S_1.x = S_2.x$

The expected tuple arrival rate of streams  $S_1$  and  $S_2$  at each node of the DHT is  $\frac{\lambda_1}{n}$  and  $\frac{\lambda_2}{n}$  respectively. Thus, the expected number of join tuples generated by  $S_1$  and  $S_2$  over sliding windows at each node can be estimated as

$$T(S_1, S_2) = sel \times \left(\frac{W_1 \lambda_1}{n}\right) \times \left(\frac{W_2 \lambda_2}{n}\right) \quad (1)$$

Each node needs a memory space for storing tuples in its local sliding window equivalent to  $\frac{W_1 \lambda_1}{n}$  and  $\frac{W_2 \lambda_2}{n}$ . In general, if  $(\frac{W_i \lambda_i}{n} > m(S_i))$  we have a loss rate (Lr) to store tuples equivalent to:

$$Lr(S_i) = \begin{cases} 0, & \frac{W_i \lambda_i}{n} \leq m(S_i) \\ \frac{W_i \lambda_i}{n} - m(S_i), & otherwise \end{cases} \quad (2)$$

Assuming that memory is insufficient to retain all the tuples in  $W_1$  and  $W_2$ , the loss of join tuples  $L$  of  $S_1$  and  $S_2$  is:

$$L(S_1) = sel \times Lr(S_1) \times \left(\frac{W_2 \lambda_2}{n}\right) \quad (3)$$

$$L(S_2) = sel \times Lr(S_2) \times \left(\frac{W_1 \lambda_1}{n}\right) \quad (4)$$

Let  $\alpha_i$  be the  $S_i$ -tuples stored in the memory space  $m(S_i)$  and  $\beta_i$  be the  $S_i$ -tuples not stored due to memory constraints (see Figure 3). We can rewrite equations (3) and (4) as:

$$L(S_1) = sel \times \beta_1 \times (\alpha_2 + \beta_2) = (sel \times \alpha_2 \times \beta_1) + (sel \times \beta_1 \times \beta_2)$$

$$L(S_2) = sel \times \beta_2 \times (\alpha_1 + \beta_1) = (sel \times \alpha_1 \times \beta_2) + (sel \times \beta_1 \times \beta_2)$$

Notice that the tuples related to expression  $(sel \times \beta_1 \times \beta_2)$  are counted in both  $L(S_1)$  and  $L(S_2)$ . This expression can be rewritten as:  $(sel \times Lr(S_1) \times Lr(S_2))$ . The total loss of join

tuples  $TL$  of  $S_1 \bowtie S_2$  is the sum of the loss of join tuples  $L(S_1)$  and  $L(S_2)$  minus the tuples counted twice:

$$TL(S_1, S_2) = L(S_1) + L(S_2) - (sel \times Lr(S_1) \times Lr(S_2)) \quad (5)$$

The completeness  $C$  of a  $S_1 \bowtie S_2$  join query is the fraction of total results  $T(S_1, S_2)$  minus the loss of tuples  $TL(S_1, S_2)$  and total results  $T(S_1, S_2)$ , using equation (1) and equation (5)  $C$  is:

$$C = \frac{T(S_1, S_2) - TL(S_1, S_2)}{T(S_1, S_2)} \quad (6)$$

Developing expressions in (6) allows us to simplify  $C$  to:

$$C = \frac{n^2 \times m(S_1) \times m(S_2)}{W_1 \lambda_1 \times W_2 \lambda_2} \quad (7)$$

Moreover, we can write (7) as:

$$n = \sqrt{\frac{C \times (W_1 \lambda_1) \times (W_2 \lambda_2)}{m(S_1) \times m(S_2)}} \quad (8)$$

This equation allow us to evaluate how many peers are necessary to process a 2-way join query.

Now we generalize our analysis to  $m$ -way joins as following. Recall that the total loss of join tuples  $TL$  is the sum of the loss of join tuples minus the tuples counted more than one time. The sum of the loss of join tuples can be easily extended to an  $m$ -way join as  $\sum_{i=1}^m L(S_i)$ . However, the expression that represents the tuples counted more than one time is more difficult to generalize. We use the same method of rewriting (3) and (4) to find the expression that represents the case of tuples counted more than one time. Thus in a  $S_1 \bowtie S_2 \bowtie S_3$  join we rewrite  $L(S_1), L(S_2)$  and  $L(S_3)$ , discovering that  $(sel^2 \times \beta_1 \times \beta_2 \times \alpha_3)$ ,  $(sel^2 \times \beta_1 \times \beta_3 \times \alpha_2)$  and  $(sel^2 \times \beta_2 \times \beta_3 \times \alpha_1)$  are counted twice and  $(sel^2 \times \beta_1 \times \beta_2 \times \beta_3)$  is counted triple. Rewriting  $\alpha_i$  and  $\beta_i$  we arrive at the following expression:  $sel^2 Lr(S_1) Lr(S_2) m(S_3) + sel^2 Lr(S_1) Lr(S_3) m(S_2) + sel^2 Lr(S_2) Lr(S_3) m(S_1) + 2 sel^2 Lr(S_1) Lr(S_2) Lr(S_3)$ .

Repeating the same method with  $m$ -way joins ( $m \geq 4$ ) and analyzing the resulting expressions, we arrive at the following general expression for a  $S_1 \bowtie S_2 \bowtie \dots \bowtie S_m$  join:

$$\sum_{k=2}^m \sum_{\substack{S' \subseteq S \\ |S'|=k}} \sum_{\substack{S'' \subseteq S \\ |S''|=m-k \\ S'' \cap S' = \emptyset}} (sel^{m-1}(k-1) \prod_{a \in S'} Lr(a) \prod_{b \in S''} m(b))$$

Now, the general case of (5) can be expressed as:

$$TL(S_1, S_2, \dots, S_m) = \sum_{i=1}^m L(S_i) - \sum_{k=2}^m \sum_{\substack{S' \subseteq S \\ |S'|=k}} \sum_{\substack{S'' \subseteq S \\ |S''|=m-k \\ S'' \cap S' = \emptyset}} (sel^{m-1}(k-1) \prod_{a \in S'} Lr(a) \prod_{b \in S''} m(b)) \quad (9)$$

The completeness  $C$  of a  $S_1 \bowtie S_2 \bowtie \dots \bowtie S_m$  join query, using the general form of (1) and equation (9) is:

$$C = \frac{T(S_1, S_2, \dots, S_m) - TL(S_1, S_2, \dots, S_m)}{T(S_1, S_2, \dots, S_m)} \quad (10)$$

Developing expressions in (10) allows us to simplify  $C$  to:

$$C = \frac{n^m \prod_{i=1}^m m(S_i)}{\prod_{i=1}^m W_i \lambda_i} \quad (11)$$

and to obtain

$$n = \sqrt[m]{\frac{C \times \prod_{i=1}^m W_i \lambda_i}{\prod_{i=1}^m m(S_i)}} \quad (12)$$

It is clear from our analysis that (11) is independent of selectivity which is reasonable in the context of continuous join queries. As our analysis shows, DHTJoin can scale up the processing of continuous join queries using multiple peers and improve the completeness of join results. Using (12) a DHTJoin user can adjust the size of the network by evaluating how many peers are necessary to process a continuous join query for given stream arrival rates and a desired result completeness.

## 6. Performance Evaluation

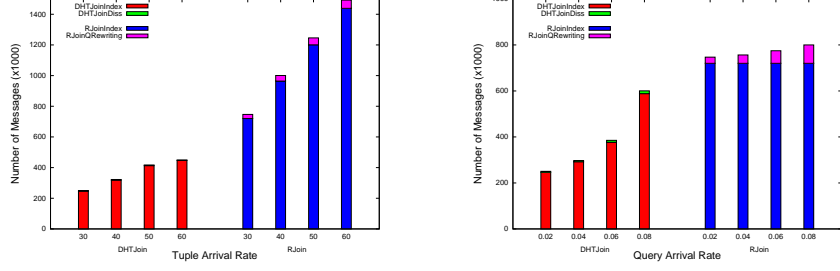
In this section, we provide an extensive performance evaluation of our method through simulation, compared with a baseline method.

**Simulator.** To test our DHTJoin method, we built a Java-based simulator, using Chord which is a simple and efficient DHT. We use a discrete event simulation package SimJava to simulate the distributed processing. The network size is set to 1024 nodes. To simulate a node, we use a Java object that performs all tasks that must be done by a node in the DHT, in the dissemination procedure and in the join query processing. In order to assess our approach, we compare the performance of DHTJoin against a complete implementation of RJoin. [14] which is the most relevant related work (see Section 7). RJoin uses incremental evaluation based on tuple indexing and query rewriting over distributed hash tables. In RJoin a new tuple is indexed twice for each attribute it has; wrt the attribute name and wrt the attribute value. A query is indexed waiting for matching tuples. Each arriving tuple that is a match causes the query to be rewritten and reindexed at a different node.

**Data generation.** We generate arbitrary input data streams consisting of synthetic asynchronous data items with no tuple-level semantics. We have a schema of 10 relations, each one with 10 attributes. In order to create a new tuple we choose a relation using an uniform distribution and assign values to all its attributes using a Zipf distribution with a default parameter of 0.9. The max value of the domain of the join attribute is fixed to 1000. Unless otherwise specified, tuples on streams are generated at a constant rate of  $\lambda_i = 30 \text{ tuples/second}$ .

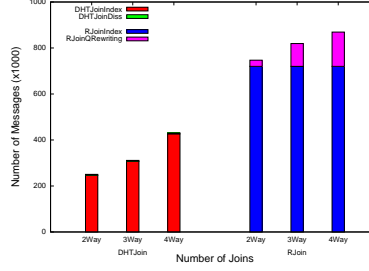
**Query generation.** Unless otherwise specified, queries are generated with a mean arrival rate of 0.02, i.e., a query arrives to the system every 50 seconds on average. We generate





(a) Effect of tuple arrival rate

(b) Effect of query arrival rate



(c) Effect of number of joins

Figure 4: Effect of tuple, query arrival rates and number of joins on the network traffic

queries of type 1 to evaluate the tuples' arrival rate and query's arrival rate. The effect of number of joins was evaluated using queries of type 2. In all experiments, we use time-based sliding windows of 50 seconds. The default duration of our experiments is 300 seconds.

In the rest of this section, we evaluate network traffic and the effectiveness of the approaches proposed in Section 4 to deal with node failures.

### 6.1. Network Traffic

In this section, we investigate the effect of tuples' arrival rate, query's arrival rate and number of joins on the network traffic. The network traffic is the total number of messages needed to index tuples and disseminate a query in DHTJoin or to index tuples and perform query rewriting in RJoin. The network traffic of RJoin and DHTJoin grows as the tuples' arrival rate grows. In RJoin, as more tuples arrive, the number of messages related to the indexing of tuples and query rewriting increases (see Figure 4(a)). DHTJoin generates significantly less messages than RJoin. The reason is that before indexing a tuple, DHTJoin checks for the existence of a query that requires it, but RJoin indexes all tuples twice (even if there is no query for them). In Figure 4(b), we show that, as more queries arrive, RJoin generates more query rewriting messages. However, DHTJoin generates more messages only if new submitted queries contain attributes not present in the set of predicates  $\mathcal{P}$  of already submitted queries. Figure 4(c) shows that more join require more network traffic. RJoin generates more query rewriting when there are more joins in the queries. However, in DHTJoin the network traffic increases only if the arriving queries require attributes that are not present in the already disseminated queries. The reason is that with the dissemination of queries,

DHTJoin can avoid the unnecessary indexing of tuples that are not required by the queries.

In summary, due to the integration of query dissemination and hash-based placement of tuples our approach avoids the excessive traffic generated by RJoin which is due to its method of indexing tuples.

## 6.2. Node Failures

We now investigate the effect of the approach proposed in Section 4.2 in order to deal with node failures during query execution. In our experiments, we repeat the same scenario of Figure 1(b) with  $\lambda_i = 400$  tuples/sec. In Figure 5, we show that, as the period of inactivity (time between fail and recovery) of a stream source gets longer, the generation of tuples that never contribute to join results increases. However, by eliminating unnecessary intermediate results, this optimization yields an important reduction of network traffic.

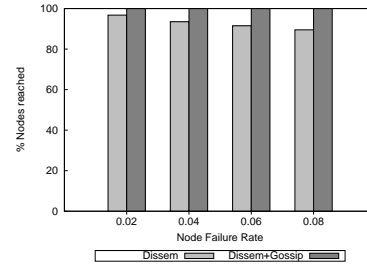
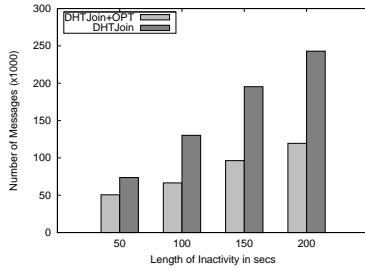


Figure 5: Reduction of intermediate results and its impact on network traffic

Figure 6: Effect of dealing with node failures during the dissemination procedure

## 6.3. Failures During Dissemination

The failure of a node in the tree structure generated by the dissemination procedure makes the entire subtree under this node unreachable. To provide reliability in the dissemination of queries, we proposed a gossip based protocol (see Section 4.1). To evaluate the effectiveness of our approach regarding an increment of node's failure rate we originate queries every 100 seconds on average and we increment the node's failure rate (see Figure 6). We consider a first scenario where the queries are disseminated using the technique described in Section 3.1 and a second scenario where the queries are disseminated using the same technique in complement with gossip. Figure 6 shows that with node failures the dissemination cannot achieve a network coverage of 100%. However, the dissemination of queries complemented with gossip can obtain a network coverage of 100% in spite of an increase in node failures.

## 7. Related Work

Unstructured P2P networks typically use a simple flooding scheme which is inefficient in terms of response time and consumes much network traffic. Furthermore, they are not suitable for efficient processing of continuous join queries as they do not provide guarantees of any kind. Structured networks (i.e. DHT) provide more efficient key-based search. Because

applications that process streams from different sources are inherently distributed and because distribution is a well accepted approach to improve both performance and scalability [8][29] of a DSMS, using a DHT is a natural choice to face the challenges motivated by the processing of continuous join queries. DHTJoin exploits the power of a DHT in two major ways. First, to disseminate queries using a tree based on the information stored in the DHT routing table. This information is maintained by the DHT protocol and does not entail any extra processing cost for DHTJoin. Second, to index tuples for query processing and detect a failure on a node that participates in query processing tasks.

A DHT can serve as the hash table that underlies many parallel hash-based join algorithms. However, our approach provides Internet-wide scalability. Our work is related to many studies in the field of centralized and distributed continuous query processing [13][10][30][6][20]. In PIER [13], a query processor is used on top of a DHT to process one-time join queries. Recent work on PIER has been developed to process only continuous aggregation queries. PeerCQ [10] was developed to process continuous queries on top of a DHT. However, PeerCQ does not consider SQL queries and the data is not stored in the DHT. Borealis [30], TelegraphCQ [6] and DCAPE [20] have been developed to process distributed continuous queries and many of their techniques for load-shedding and load balancing are orthogonal to our work. In Seaweed [21] a scalable query infrastructure built on top of a DHT to process one-shot queries rather than continuous queries. However, Seaweed does not use the DHT to distribute data but to replicate metadata and to disseminate queries. An algorithm for supporting ranked join queries in P2P networks was introduced in [34]. Irrelevant top-k tuples are pruned of local nodes before they are sent to be probed for join matches. However, this work does not consider continuous queries. The most relevant previous work regarding the utilization of a DHT network to process continuous queries is [14] which proposes RJoin, an algorithm that uses incremental evaluation. This incremental evaluation is based on tuple indexing and query rewriting over distributed hash tables. A major difference in our work differs is that DHTJoin avoids indexing tuples that cannot contribute to generate join results and deals with the dynamic behaviour of peers.

To disseminate a query, DHTJoin dynamically builds a dissemination tree as proposed in [9]. However, this work does not consider the dynamic behaviour of nodes. To solve this problem, we propose a gossip based solution that considers the utilisation of the neighbor list to provide fault tolerance. The probabilistic dissemination algorithm named Randcast proposed in [17] spreads messages very fast but fails to reach every node in the network. The protocol proposed in [18] assures a good tradeoff between message overhead and reliability guarantee using a specific connection graph. However, its main drawback is the maintenance of such graph that requires global knowledge of membership. In our work, the structure that supports the membership protocol is supported by the DHT and does not require global knowledge of membership for its maintenance.

The notion of result completeness has been studied in the context of P2P databases. A solution to estimate the completeness has been proposed in [16] for one-time queries. Completeness is computed at the peer level using the notion of routing graphs. The routing graphs trace the routes that a one-time query and its sub-queries take through the network. In the Seaweed query infrastructure [21], data summaries and availability models are used in order to predict query completeness and response times to one-shot queries. Our work instead considers continuous queries and completeness is calculated on a data level not considering

data summaries.

## 8. Conclusion

In this paper, we proposed a new method, called DHTJoin, for processing continuous join queries using DHTs. DHTJoin combines hash-based placement of tuples and dissemination of queries using the trees formed by the underlying DHT links. DHTJoin takes advantage of the indexing power of DHT protocols and dissemination of queries to avoid the placement of tuples that cannot contribute to generate join results. We showed analytically that DHTJoin can scale up the processing of continuous join queries using multiple peers and improves the completeness of join results linearly as the memory capacity is increased.

To validate our contribution, we implemented DHTJoin as well as RJoin which is the most relevant state of the art solution in the context of processing continuous join queries using DHTs. Our performance evaluation shows that DHTJoin yields significant performance gains due to the mechanisms of indexing tuples and the elimination of unnecessary intermediate results. Our results also demonstrate that the total number of messages of DHTJoin is always less than that of RJoin wrt tuple arrival rate, query arrival rate and number of joins. We show that the problem of node failures during the dissemination of queries can be complemented with a gossip based protocol that allows, in spite of node failures, a network coverage of 100%. We also showed that our approach to deal with node failures during query execution prevents nodes of sending intermediate results that do not contribute to join results, thereby reducing network traffic.

As future work, we plan to address the problem of efficient execution of top-k join queries over data streams using DHTs, taking advantage of the best position algorithms [1] which can be used in many distributed and P2P systems for efficient processing of top-k queries.

## References

- [1] R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *VLDB*, pages 495–506, 2007.
- [2] A. Arasu and J. Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*, 33(3):6–12, 2004.
- [3] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *SIGMOD Conference*, pages 515–526, 2004.
- [4] P. Bonnet et al. Towards sensor database systems. In *Mobile Data Management*, pages 3–14, 2001.
- [5] M. Castro et al. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *INFOCOM*, 2003.
- [6] S. Chandrasekaran et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
- [7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaraqc: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.
- [8] G. Cormode and M. N. Garofalakis. Streaming in a connected world: querying and tracking distributed data streams. In *SIGMOD Conference*, pages 1178–1181, 2007.
- [9] S. El-Ansary et al. Efficient broadcast in structured p2p networks. In *IPTPS*, pages 304–314, 2003.
- [10] B. Gedik and L. Liu. Peercq: A decentralized and self-configuring peer-to-peer information monitoring system. In *ICDCS*, pages 490–499, 2003.
- [11] L. Golab et al. Optimizing away joins on data streams. In *SSPS*, pages 48–57, 2008.

- [12] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.
- [13] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, pages 321–332, 2003.
- [14] S. Idreos, E. Liarou, and M. Koubarakis. Continuous multi-way joins over distributed hash tables. In *EDBT*, pages 594–605, 2008.
- [15] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.
- [16] M. Karnstedt et al. Estimating the number of answers with guarantees for structured queries in p2p databases. In *CIKM*, pages 1407–1408, 2008.
- [17] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):248–258, 2003.
- [18] M.-J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministically constrained flooding on small networks. In *DISC*, pages 253–267, 2000.
- [19] B. Liu and E. A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, pages 829–840, 2005.
- [20] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, pages 1338–1341, 2005.
- [21] D. Narayanan, A. Donnelly, R. Mortier, and A. I. T. Rowstron. Delay aware querying with seaweed. *VLDB J.*, 17(2):315–331, 2008.
- [22] F. Naumann, J. C. Freytag, and U. Leser. Completeness of integrated information sources. *Inf. Syst.*, 29(7):583–615, 2004.
- [23] W. Palma, R. Akbarinia, E. Pacitti, and P. Valduriez. Efficient processing of continuous join queries using distributed hash tables. In *Euro-Par*, pages 632–641, 2008.
- [24] S. Ratnasamy et al. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.
- [25] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.
- [26] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.
- [27] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.
- [28] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, page 594, 1996.
- [29] N. Tatbul, U. Çetintemel, and S. B. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *VLDB*, pages 159–170, 2007.
- [30] N. Tatbul and S. B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006.
- [31] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.
- [32] Y. Yang and D. Papadias. Just-in-time processing of continuous queries. In *ICDE*, pages 1150–1159, 2008.
- [33] B. Y. Zhao et al. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.
- [34] K. Zhao et al. Supporting ranked join in peer-to-peer networks. In *DEXA Workshops*, pages 796–800, 2005.
- [35] Y. Zhou et al. Pmjoin: Optimizing distributed multi-way stream joins by stream partitioning. In *DASFAA*, pages 325–341, 2006.